The XML approach for the DAQ initialization files

(plus update on the status of the DAQ project)

G. Chiodini, S. Magni, D. Menasce,
L. Uplegger, D. Zhang

The DAQ initialization files

- A DAQ system gathers information for initialization of hardware and for appropriately **start/stop/resume** a run from a set of files containing the relevant data. Usually these are plain ASCII files.

- There are two possible approaches to the problem of making the data contained in an initialization file available to a program that needs them to address hardware components:

  ❶ the initialization file is just a collection of lines containing characters: ⇒ needs a set of *ad-hoc* lines of code to read them in memory (validation of syntax is also written *ad-hoc*)

  ❷ the initialization file is an **xml** file, with an associated **dtd** (data-translation-dictionary). Parsing and validation is accomplished by methods available in the **xml** parser library

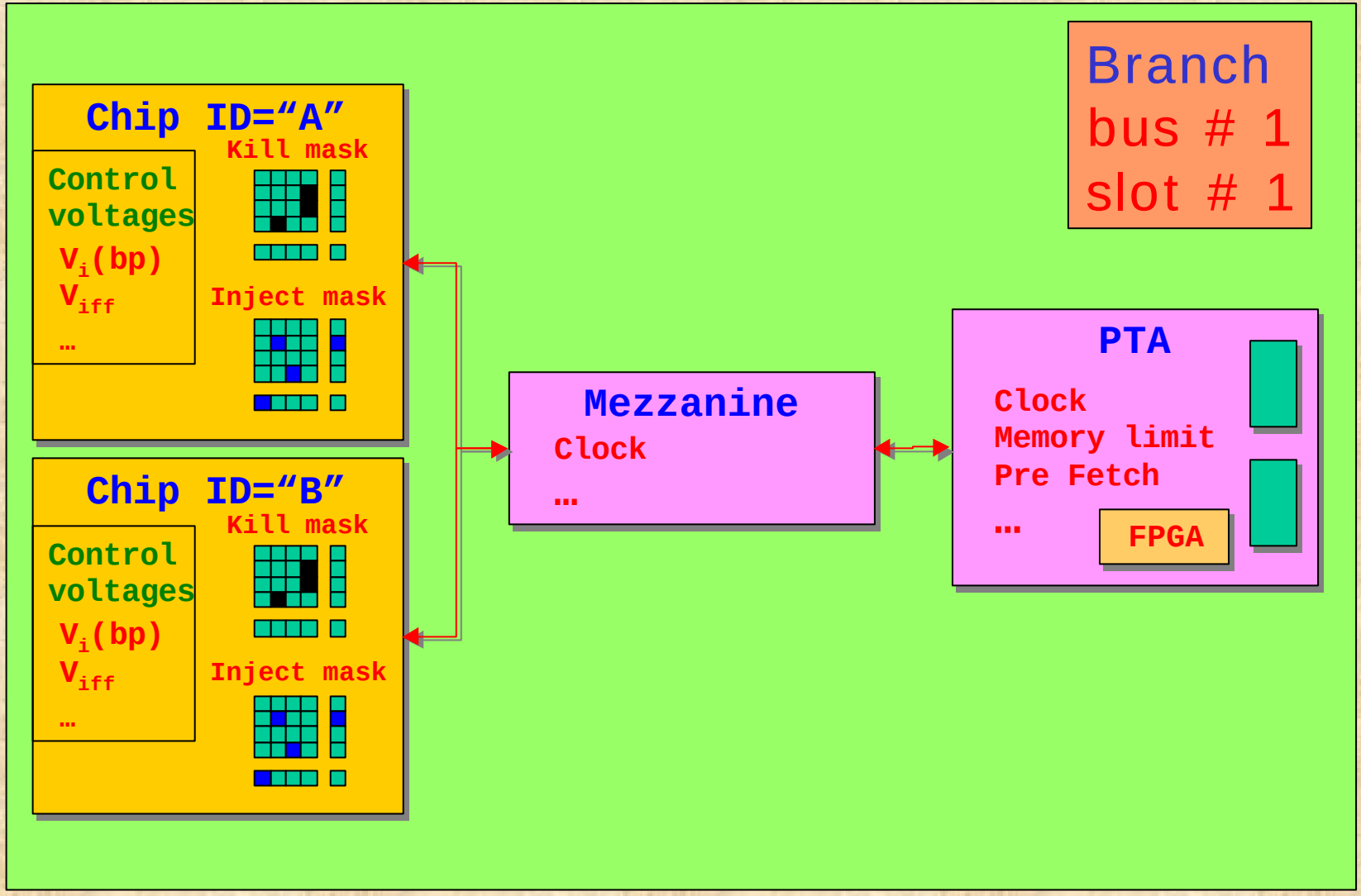
The beam test initialization file I

- In the spirit of taking advantage of this test beam to approach new software technologies, to have time to learn their strengths and weaknesses and, at the same time, to provide flexible tools for the DAQ itself, we have chosen to adopt the second strategy:

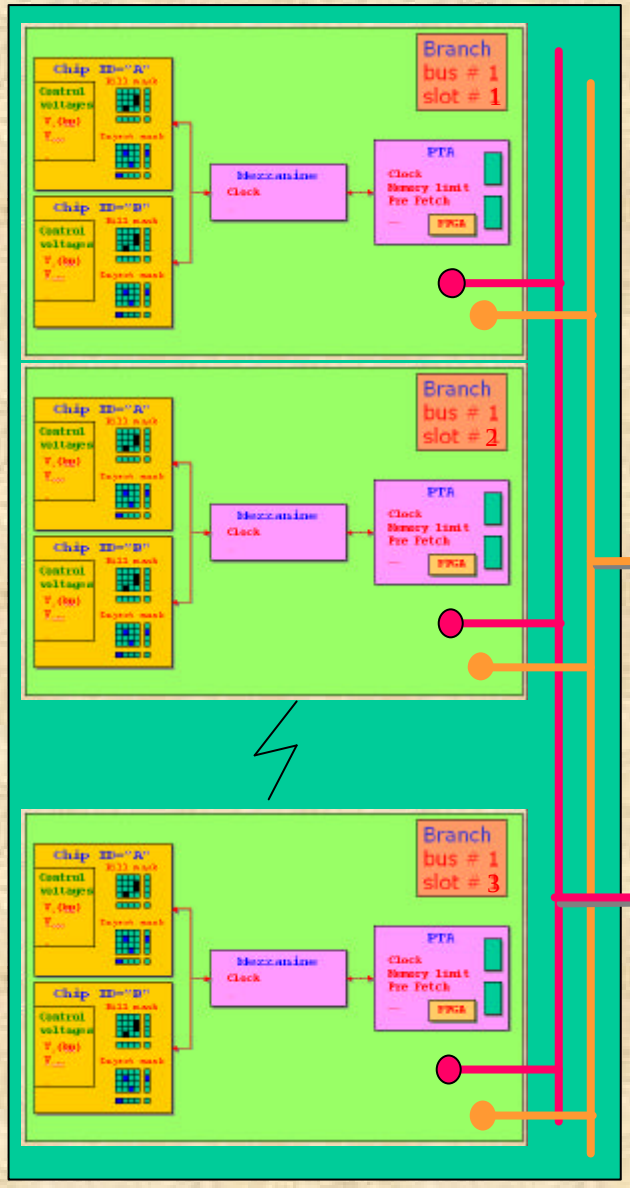
  we have defined an xml syntax that models our hardware and make use of a parser/validator to gather data in memory.

- First step to this goal is the design of the elements required to be present in the initialization file: they should reflect what the DAQ must be knowledgeable about (hardware components, their status, their mutual relationship and hierarchy,…)

The beam test initialization file II



The beam test initialization file III



Initialization File (XML)

Pulser
(GPIB device)

High/Low Voltage CAEN Power Supply
(TCP-IP device)

Structure of the XML file

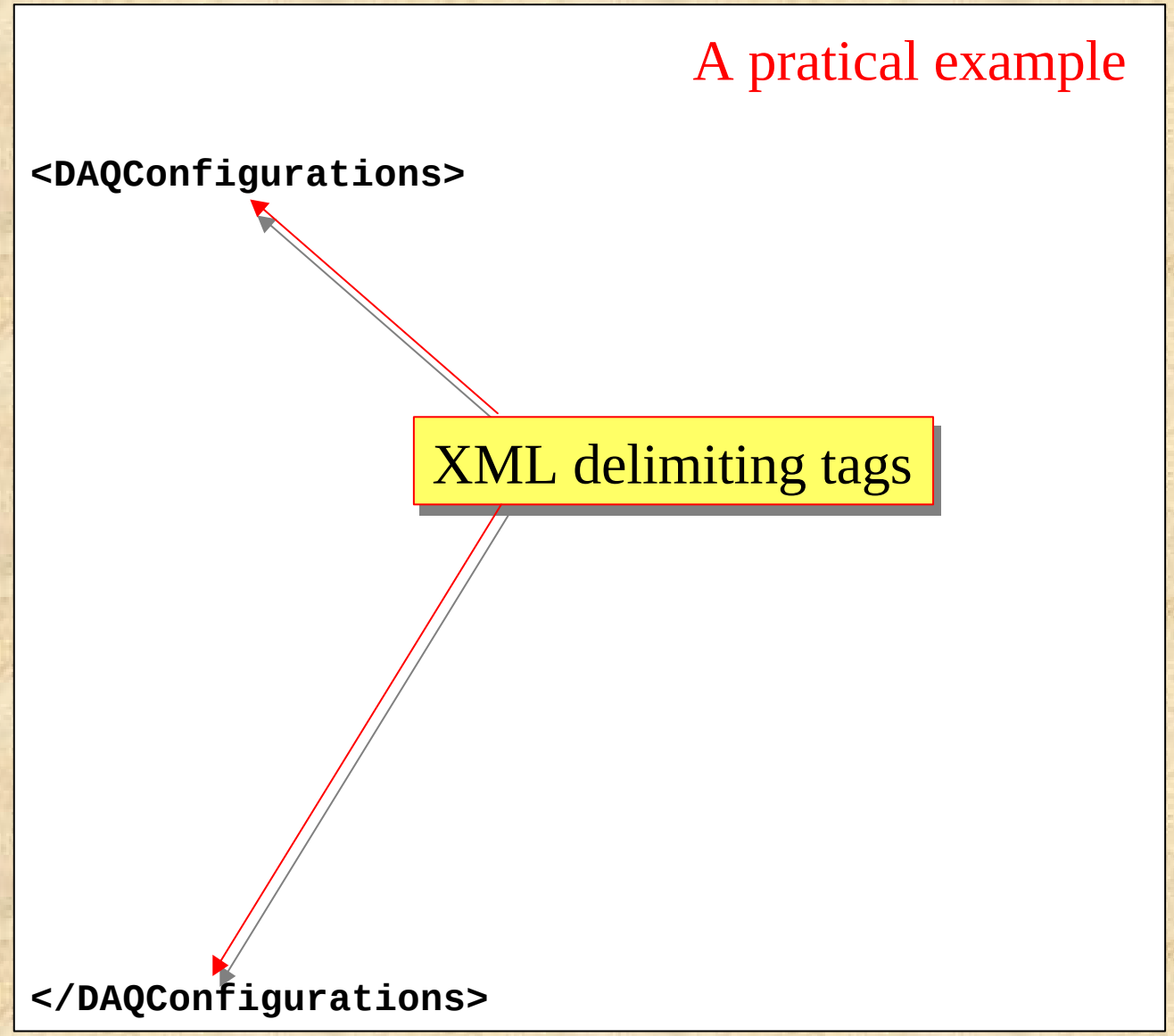
- The XML initialization file has been designed to closely resemble the hardware it is supposed to represent.

- Each hardware component is described by an embracing pair of XML tags with a clearly laid out structure. Each tag may also have qualifiers attached to it to further specify details of the particular equipment referenced.

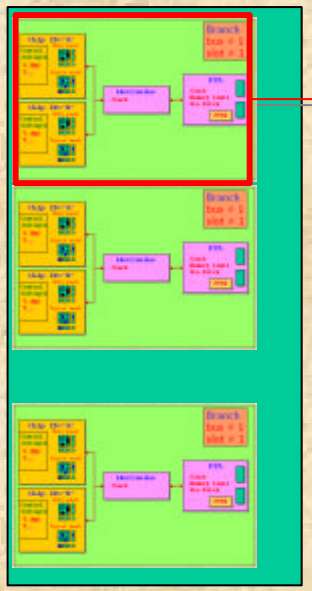
Let’s step through an example which describes our approach:

Structure of the XML file

A pratical example



Structure of the XML file



```
<DAQConfigurations>

<Branch Bus="1" Slot="1" Status="1" Comment="">

</Branch>

</DAQConfigurations>
```

Structure of the XML file

```
<DAQConfigurations>
  <Branch Bus="1" Slot="1" Status="1" Comment="">
  </Branch>
  <Branch Bus="2" Slot="2" Status="1" Comment="">
  </Branch>
</DAQConfigurations>
```

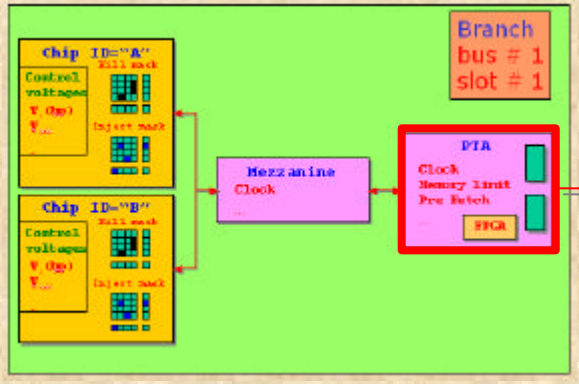
Structure of the XML file

Pulse Generator

Power Supply

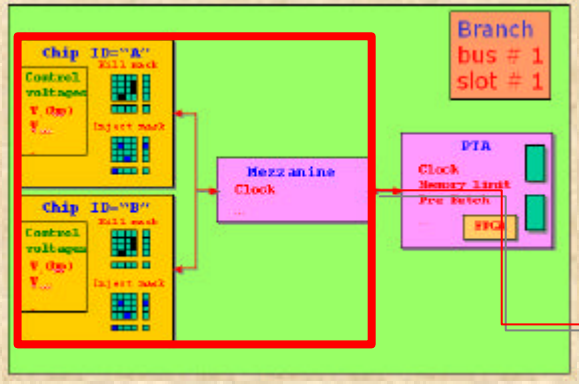
```
<DAQConfigurations>

  <Branch Bus="1" Slot="i" Status="1" Comment="">
    ……..
  </Branch>

  <Branch Bus="2" Slot="j" Status="1" Comment="">
    ……..
  </Branch>

  <GPIB Version="0.0.1" Comment="PulseGenerator">

  </GPIB>

  <CAEN Version="0.0.1" Comment="PowerSupply">

  </CAEN>

</DAQConfigurations>
```

Structure of the XML file



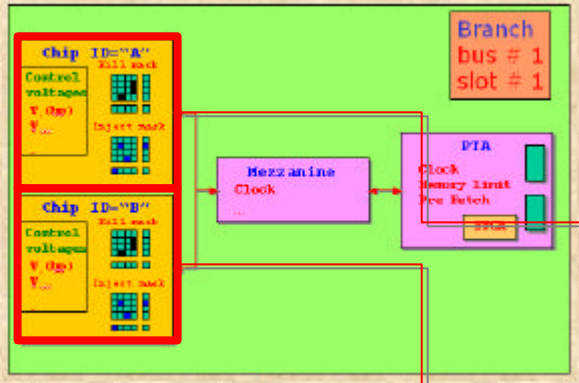

```
<DAQConfigurations>
 <Branch Bus="1" Slot="1" Status="1" Comment="">

  <PTA DataSource="Mezzanine" Comment="Fake">
   ..............................
  </PTA>

 </Branch>
</DAQConfigurations>
```

Structure of the XML file



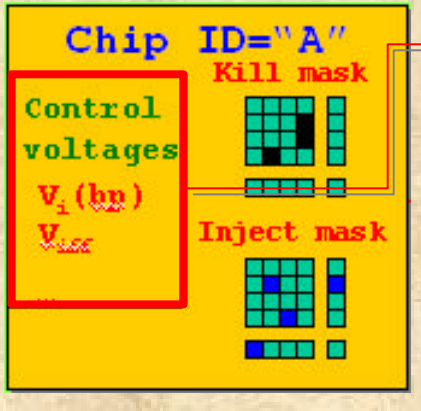

```
<DAQConfigurations>
 <Branch Bus="1" Slot="1" Status="1" Comment="">

  <PTA DataSource="Mezzanine" Comment="Fake">
   ..............................
  </PTA>

  <Mezzanine Comment="">









  </Mezzanine>

 </Branch>
</DAQConfigurations>
```

Structure of the XML file



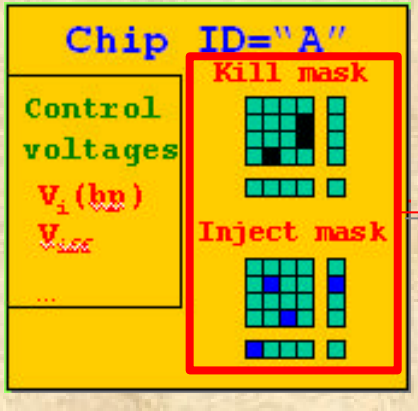

```
<DAQConfigurations>
 <Branch Bus="1" Slot="1" Status="1" Comment="">

  <PTA DataSource="Mezzanine" Comment="Fake">
   ..............................
  </PTA>

  <Mezzanine Comment="">
   <Detector ID="A" Type="preFPIX2" Comment="">
    ..............................
   </Detector>

   <Detector ID="B" Type="preFPIX2" Comment="">
    ..............................
   </Detector>
  </Mezzanine>

 </Branch>
</DAQConfigurations>
```

Structure of the XML file




```
<DAQConfigurations>
 <Branch Bus="1" Slot="1" Status="1" Comment="">
  ..............................
  <Mezzanine Comment="">
   <Detector ID="A" Type="preFPIX2" Comment="">
    <Vipb Value="120" Comment=""/>
    <Vtho Value="60"  Comment=""/>
    ..................

      ..................
      ..................

   </Detector>
  </Mezzanine>

 </Branch>
</DAQConfigurations>
```

Structure of the XML file




```
<DAQConfigurations>
 <Branch Bus="1" Slot="1" Status="1" Comment="">
  ..............................
  <Mezzanine Comment="">
   <Detector ID="A" Type="preFPIX2" Comment="">
    <Vipb Value="120" Comment=""/>
    <Vtho Value="60"  Comment=""/>
    ..................
    <Cell Row="15" Col="4"  Kill="y" Inj="y">
    <Cell Row="18" Col="14" Kill="y" Inj="y">

     ..................
     ..................

   </Detector>
  </Mezzanine>

 </Branch>
</DAQConfiguration>
```

Structure of the XML file

The process of definition goes on until all the details of the detector and its associated electronics have been laid out.

The next step is the definition of a validation-dictionary to specify which field is required, which is optional, how many elements at least are necessary, which is the relative hierarchy and so on and so forth. Such a dictionary is a **dtd**-file.

The syntax of a **dtd-file** is specified by the XML protocol (available at http://www.w3.org/TR/REC-xml).

In a dtd-file you specify the syntax of your xml file

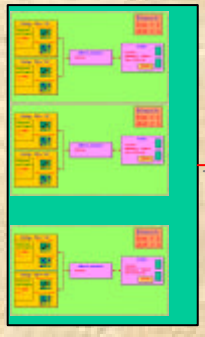
Structure of the DTD file

How does an **xml** file specify its **dtd** definition?

DAQcfg.xml

```
<!DOCTYPE DAQConfigurations SYSTEM “DAQcfg.dtd”>


<DAQConfigurations>
 <Branch Bus=“1” Slot=“1” Status=“1” Comment=“”>
  ……
  <Mezzanine Comment=“”>
   <Detector ID=“A” Type=“preFPIX2” Comment=“”>

   </Detector>
  </Mezzanine>

 </Branch>
</DAQConfiguration>
```

Structure of the DTD file



```
<!ELEMENT DAQConfigurations (DAQ, GPIB, CAEN) >
```

Pulse Generator

Power Supply

Structure of the DTD file

DAQcfg.dtd

```
<!ELEMENT DAQConfigurations (DAQ, GPIB, CAEN) >
```

DAQcfg.xml

```
<!DOCTYPE DAQConfigurations SYSTEM "DAQcfg.dtd">
<DAQConfigurations>

  <DAQ Version="0.0.1" Comment="">
    ……..
  </DAQ>

  <GPIB Version="0.0.1" Comment="PulseGenerator">

  </GPIB>

  <CAEN Version="0.0.1" Comment="PowerSupply">

  </CAEN>

</DAQConfigurations>
```

Structure of the DTD file

DAQcfg.dtd

```
<!ELEMENT DAQConfigurations (DAQ, GPIB, CAEN) >

<!ELEMENT DAQ ( DaqHome,
                LogFile,
                DaqMode,
                FakeData*,
                Branch+ ) >
```

Optional field

More than one instance allowed (but at least one)

DAQcfg.xml

```
<!DOCTYPE DAQConfigurations SYSTEM "DAQcfg.dtd">
<DAQConfigurations>

 <DAQ Version="0.0.1" Comment="">
   <DaqHome Value="…" Comment="…" />
   <LogFile Value="…" Comment="…" />
   <DaqMode Value="…" Comment="…" />
   <FakeData Interspill="10" … />
   <Branch Bus="…" Slot="…">
   </Branch>
   <Branch Bus="…" Slot="…">
   </Branch>
 </DAQ>
</DAQConfigurations>
```

Structure of the DTD file

DAQcfg.dtd

```
<!ELEMENT DAQConfigurations (DAQ, GPIB, CAEN) >

<!ELEMENT DAQ ( DaqHome,
                LogFile,
                DaqMode,
                FakeData*,
                Branch+ ) >
<!ATTLIST DAQ
   Version CDATA #REQUIRED
   Comment CDATA #IMPLIED
>
```

DAQcfg.xml

```
<!DOCTYPE DAQConfigurations SYSTEM "DAQcfg.dtd">
<DAQConfigurations>

 <DAQ Version="0.0.1" Comment="Under development">
  ……….
 </DAQ>

</DAQConfigurations>
```

Parsing and validation

Once a DTD has been defined and a possible instance of an XML file (based on the chosen DTD) has been implemented, we already have two important benefits:

- People working on the GUI to manage the xml file communicate with people using the xml in the DAQ part just by means of the **DTD** definition. Should a new field be required, with specific fields and sub-fields, all is needed in order to share this info among the parties involved in the code development is just the **DTD** file itself, which uniquely defines what is and what is not allowed to be present in an xml file based on that particular grammar.

Code development is made easier
The DTD already represents the full documentation needed

Parsing and validation

- The order in which fields are declared in the xml file is irrelevant: the parser automatically transfers fields and associated values in memory. This makes easy to add/remove elements in the configuration file without modifying the code.

  More important, errors in the xml file are detected by the parser/validator with no need to stuff the code with consistency checks (the developer of the code is relieved from the tedious task of providing consistency check all along).

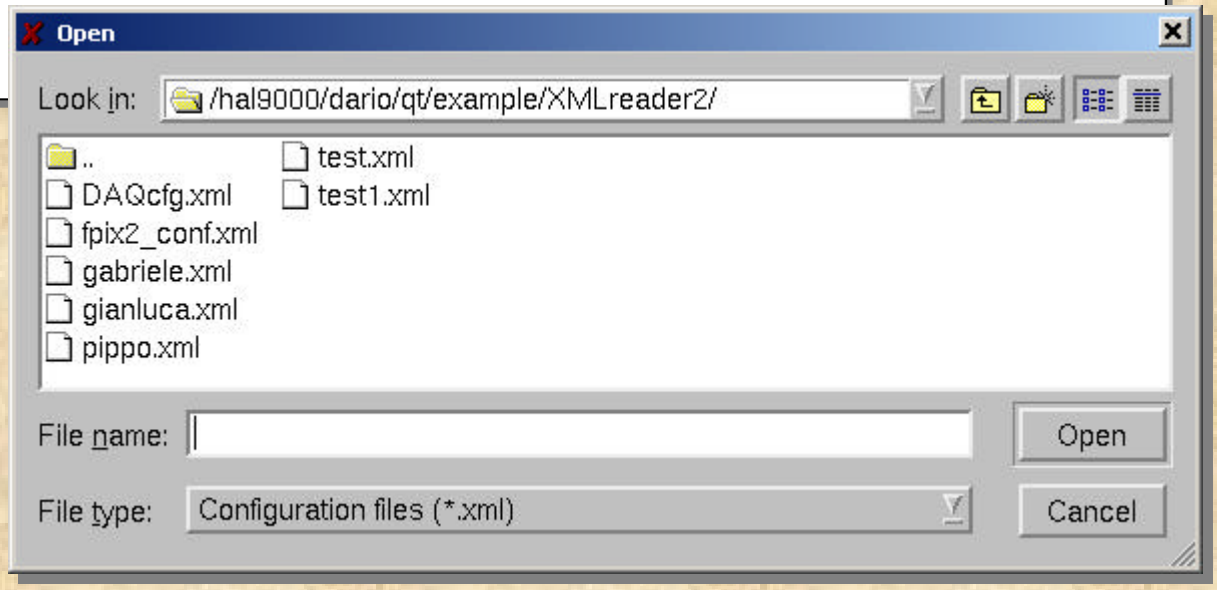
The parser/validator we decided to use is xerces, available at http://xml.apache.org/xerces-c (as usual, it is public domain software, which easily integrates with Qt and JungoDriver).

The Qt library also provides a parser, but the current version has no validation features, so we decided to use xerces

Integration with the DAQ

- Let's see how the parsing is actually implemented: open a dialog box to navigate the directory tree and select an xml file

```
 ...
CfgFile = QFileDialog::getOpenFileName( QString::null,
                                        "Configuration files (*.xml)",
                                        this );
if (!CfgFile) {return;}
 ...
```



Integration with the DAQ

- Once the xml file is chosen, let's open it, parse and validate

Enable validation capabilities

```
DOMParser parser;
parser.setValidationScheme(DOMParser::Val_Always);

DOMTreeErrorReporter *errReporter = new DOMTreeErrorReporter();
parser.setErrorHandler(errReporter);


parser.parse(DAQConfigurator::CfgFile.c_str());


for ( std::vector<C_Branch*>::iterator  i=DAQ->Branch.begin();
                                       i!=DAQ->Branch.end();
                                       i++ ) {
 cout << "DAQ->Branch->Mezzanine->Detectors[0]->Vibp->Value = " <<
       (*i)->Mezzanine->Detectors[0]->Vibp ->Value << endl;
}
 ...
```

Specify a class to handle generated errors

Do parsing AND validation

Use variables read from xml

Results

- Once we defined the dtd for the initialization files and created an instance of an xml file, we tried to actually initialize and read a real detector.
- We did this remotely on the computer at Feynman using a preFPIX2Tb pixel device. After some bug-hunting we were able to perform a complete initialize/read-back cycle.
- At a certain point, unfortunately, we burned the device, so at this point we are deprived of the possibility to continue our tests since this was the only device available to us.

This is a problem, since also code development becomes difficult if one cannot test parts as soon as they are supposed to work.

A possible way out is to concentrate on an FPIX1 (we have one in Milano), but in general it would be wise to have at least two (or better three) devices of each kind ready for testing in the not to far future...

To do

We think we have made a new significant step forward:

- Code to read/validate an xml initialization file exist and works properly (a utility class has been developed to this extent)
- A full cycle of initialization/read–back has been shown to work properly on a preFPIX2Tb detector
- This has been accomplished within the general framework we have developed (cooperating tasks accessing a shared memory)

Next steps

- Cooperate with Dehong Zhang on the xml editor (GUI)
- Try the same exercise with an FPIX1 detector (we need Brad to provide the necessary firmware for the mezzanine)
- Start working on auxiliary software to control external power supply and pulse generator (already have a working prototype)

To do

Long range planning

- We still have not tried to write down events on disk: should be easy, but we have to do it…
- Event builder works, but we have not yet had the chance to test it under real conditions: to do this we need at least three working devices…
- Still no real thoughts about management of initialization files history (database, WEB interface and such…)
- We have a program that does calibrations (old Gabriele's stuff): we need to re-implement it in the new framework (rather big project) and make it general enough to accommodate functionality for the different brands of pixel devices.

To do

Long range planning (contd.)

- Almost no documentation has been provided yet
- Still no planning whatsoever of integration with other detectors. Keep in mind that our mission can be stated as follows:

  provide code to read out a PTA card, regardless of what it may contain at any given time; aside from this, provide specific code to initialize and calibrate pixels and control ancillary hardware components

- If a detector has the necessary equipment (mezzanine and corresponding firmware) to fill the PTA memories (with time-stamped data), integration is straightforward.